An Architecture for Mostly Functional Languages

Tom Knight

Symbolics, Inc.
and
The M.I.T. Artificial Intelligence Laboratory
Cambridge, Massachusetts

Abstract

The addition of small amounts of special purpose hardware to conventional machines increases the efficiency of Lisp systems with dynamic type checks. In a similar way, this paper proposes additional hardware to dynamically check the correctness of parallel execution of Lisp programs. The key idea is the use of fully associative caches as a means of maintaining and enforcing a set of dependencies between portions of the program.

Introduction

This paper describes a computer architecture characterized by its ability to execute several portions of a program in parallel, while giving the appearance to the programmer that the program is being executed sequentially. This automatic extraction of parallelism makes the architecture attractive because we can exploit the existing programming tools, languages, and algorithms which have been developed over the past several decades of computer science research, while we gradually endeavor to improve them for the parallel environment. Not all of the benefits of parallel execution will be attained with this architecture, but the ability to extract even modest amounts of parallel execution from existing programs may be worth the effort. We will also discuss how the architecture can be extended, as algorithms are improved, to provide some explicit and modular programmer control over the parallelism.

Functional Languages

Many parallel architecture papers start by defining the programming model they support as strictly functional. Computer architects find that the strictly functional approach makes their task easier, because the order in which computation is performed, once the basic dependencies are satisfied, is irrelevant. Building highly parallel machines which execute strictly functional languages then becomes relatively straightforward. Strictly functional languages certainly have advantages [1], including the relative ease of proving their correctness and the ease of thinking and reasoning about certain simple kinds of programs.

Functional Languages Have Problems

Unfortunately, there is a class of algorithms for which the strictly functional programming style appears to be inadequate. The problems are rooted in the difficulty in interfacing a strictly functional program to a non-functional outside environment. It is impossible to program a task such as an operating system, for instance, in the unmodified strictly functional style. Clever modifications of the strict functional style have been proposed which cure this defect. For example, McCarthy's AMB operator, which returns whichever of two inputs are ready first, appears to be adequate to solve this and similar problems. Several equivalent operators have been proposed [2].

However, the introduction of such non-functional operators into the programming language destroys many of its advantages. Architects can no longer ignore the order of execution of functions. Proving program correctness again becomes a very much more difficult task. Users also find that programming with such operators is tedious and error prone. In particular, as shown by Agha [3], the presence of such an operator allows the definition of a new data type, the cell, which can be side effected in just as normal memory locations in non-functional languages. Thus, the introduction of the the AMB operator, while adequate to solve the lack of side effect problem, re-introduces many of the problems which the functional language advocates are attempting to avoid.

As Minsky says, it appears we have a choice: we can either define weak systems, such as purely functional languages, about which we can prove theorems, or we can define strong systems which we can prove little about.

Functional Languages Can Be Hard to Program

Much more serious, in my mind, than any of the theoretical difficulties of strictly functional languages, is the difficulty programmers face in implementing certain types of programs. It is often more natural to think about an algorithm in terms of the internal state of a set of objects (See, for example, chapter 3 of Abelson and Sussman [4]). This object oriented viewpoint in programming language design is incompatible with the strictly functional approach to computation. The same algorithms can be implemented in either style, and both systems are surely universal – but the fact that programmers find one representation for programs easier to think about, easier to design, and easier to

debug is, in itself, a powerful motivation for a programming language to provide that representation.

If this were merely a representation issue, there would be hope that suitable compiler technology could eventually lead to a programming language which provides support for side effects, but which compiles into purely functional operations. As we saw above, however, in the general case this is not possible.

An Alternative - Mostly Functional Languages

Side effects cause problems. Architects find it difficult to build parallel architectures for supporting them. Verification software finds such programs difficult to reason about. And programming styles, with an abundance of side effects, are difficult to understand, modify, and maintain.

Instead of completely eliminating side effects, we propose that their use be severely curtailed. Most side effects in conventional programming languages like Fortran are gratuitous. They are not solving difficult multi-tasking communications problems. Nor are they improving the large scale modularity or clarity of the program. Instead, they are used for trivial purposes such as updating a loop index variable.

Elimination of these unnecessary side effects can make code more readable, more maintainable, and, as we will argue later in this paper, faster to execute.

The Multi-Lisp Approach

Many of these same issues motivate Halstead's Multi-Lisp, a parallel variant of Scheme, [5]. The approach Multi-Lisp takes to providing access to parallel computation is the addition of programmer visible primitives to the language. The three primitives which distinguish it from conventional Lisp are:

The *future* primitive allows an encapsulated value to be evaluated, while simultaneously returning to its caller a promise for that value. The caller can continue to compute with this returned object, incorporating it into data structures, and passing it to other functions. Only when the value is examined is it necessary for the parallel computation of its value to complete.

The *pcall* primitive allows the parallel evaluation of arguments to a function. Halstead shows in his paper that pcall can be implemented as a simple macro which expands into a sequence of futures. It can be thought of as syntactic sugar for a stylized use of futures.

The *delay* primitive allows a programmer to specify that a particular computation be delayed until the result is needed. Similar in some respects to a future, delay returns a promise to compute the value, but does not begin computation of a value until the result is needed. Thus the delay primitive is not a source of parallelism in the language. It is a way of providing lazy evaluation semantics to the programmer.

Both *delay* and *future* result in an order of execution different from applicative order computation. In the absence of side effects, both will result in the same value as the equivalent program without the primitives, since they affect only the order in which the computation is performed. (This is not strictly true for *delay* since its careful use can allow otherwise non-terminating programs to return a value.)

In the presence of side effects, it is difficult to predict the behavior of a future, since its value may be computed in parallel with other computations.

While the value of a delay is deterministic, since it does not introduce additional parallelism, its time of computation is dependent on when its value is first examined. This can be very non-intuitive and difficult to think about while writing programs.

Halstead implements the future and delay primitives by returning to the caller an object of data-type *future*. This object contains a pointer to a slot, which will eventually contain the value returned by the promised computation. The future may be stored in data structures and passed as a value. Computations which attempt to reference the value of the future prior to its computation are suspended. When the promised computation completes, the returned value is stored in the specified slot, and the future becomes determined. This allows any pending suspended procedures waiting for this value to run, and any further references to the future will simply return the now computed value.

Future as a Declaration

The use of future or delay can be thought of as a declaration. Their use declares that either the computation done within the scope of the delay or future has no side effects, or that, if it does, the order in which those side effects are done relative to other computations is irrelevant. We also guarantee with such a declaration that no free or shared variable referenced by the computation is side effected by some *other* parallel-executing portion of the program.

Like all declarations, the use of future or delay is a very strong assertion. In a way similar to type declarations, their use is difficult to check automatically, is error prone, has a significant performance impact if omitted, and may function correctly for many test cases, but fail unexpectedly on others.

Advocates of strong type checking in compiled languages have attempted for years to build compilers capable of proving the type correctness of programs. We believe that they have failed. All languages sophisticated enough for serious programming require at least some dynamic checks for type safety at execution time. These checks are implemented as additional instructions on conventional architectures, or as part of the normal instruction execution sequence on more recent architectures [6]. One alternative, declaring the types and relying on the word of the programmer, while leading to good performance on conventional architectures, is dangerous, error-prone, and inappropriate for sophisticated modern programming environments.

Just as hardware has provided important runtime support for type checking, we propose the use of hardware in the runtime checking of future declarations.

Dynamic Side Effect Checks

In our proposed Lisp implementation, we do not use explicit language extensions such as future and pcall. Instead, we will assume that each evaluation is encapsulated in a future and that each call can freely evaluate its arguments in parallel. Our approach is to detect and correct those cases in which this assumption is unjustified.

Our compiler transforms the program into sequences of compiled primitive instructions, called *transaction blocks*. Each transaction block has the following characteristics:

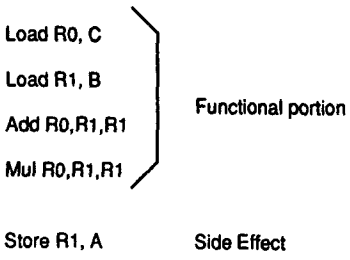


Figure 1 A Transaction Block

- Each block is independent of previous execution, except for the contents of main memory. In particular, no registers or other internal bits in the processor are shared across blocks.
- Each block contains exactly one side-effecting store into main memory, occurring at the end of the block.

Each transaction block is, from the standpoint of the memory system, a strictly functional program, consisting of register loads and register-to-register instructions, followed by a single side effecting store into main memory.

Each of these blocks can be executed independently, potentially on several different processors. Except for the relative timing of the single side-effect at the end of each block, there is no interaction between blocks. However, the order in which we execute the single side effect per block is critical. These side effects must be executed in a well defined order: the order specified in the program. Further, the execution of a side effect can potentially modify a location which some other block has already read.

These two tasks - execution of the side effects in order, and enforcing the dependencies of later blocks on earlier side effects - require special hardware.

Each processor is free to execute its block at will, up until the final side effect. Upon reaching the side effect, execution halts. Since this program is functional, this partial execution cannot affect other processors. A *block counter* is maintained, similar in purpose to a normal program counter. The block counter's job is to keep track of which block is next allowed to perform its side effect. As the block counter reaches each block in turn, the block is allowed to actually perform the side effect specified by its final instruction. This process is called *confirming* the block. Confirming a block modifies the contents of main memory. Other blocks, not yet confirmed, may already have referenced the location which has just been modified. If they have, then the data upon which they are computing is now invalid.

But the program which they are executing is functional - it has not modified main memory. We are free to abandon the partially executed block, and to attempt re-executing it a second time. This process of abandoning a partially executed block is called *aborting* the block.

In order to detect when a block needs to be aborted, we need to keep track of which memory locations it has referenced. A *dependency list*, is maintained, containing the addresses of all locations in main memory which have been referenced during execution of a block. When a processor executes a load instruction, it adds the address being loaded to the executing block's dependency list.

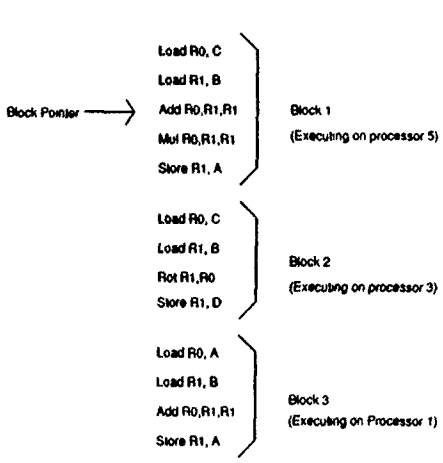


Figure 2. Several Transaction Blocks Executing in Parallel The third block will abort when the first commits, because it has referenced the variable A, which has been side effected.

When a block is confirmed, the address it is side effecting is broadcast to all other executing processors. Each processor checks its dependency list to determine if the block it is executing has referenced the modified location. If it has, the block is aborted and re-executed.

Eventually, a given block will be reached by the block counter, and it will be allowed to complete and perform its side-effect. The block counter advances to the next block, and if the block is ready, we confirm that block in the following cycle. Ideally, we will confirm a block per clock cycle. If the confirmation of one block aborts the execution of successor blocks, however, the performance will be limited by the time it takes the aborted block to re-execute its program.

The performance improvement resulting from this technique depends on two factors: the average block length, and the frequency of aborts. The longer the block, the fewer confirms are necessary to execute a given program. The more frequent the aborts, the more the block counter must await a sequential computation before confirming the next block.

We can see now the influence of reducing the number of side effects in the source program: as the number of side effects is reduced, the length of the blocks increases. Since the blocks are longer, fewer blocks need be executed to perform a computation. Since the block execution is sequential, at a rate limited to a maximum of one per clock cycle, this limits the performance of the architecture. There is probably a maximum desirable block length, since as the block size increases, the length of the dependency list, and thus the likelihood of a side effect from another block influencing the computation goes up. We are currently studying these issues in simulation, but have no results as yet.

The Order of Block Execution

Since we are simulating the behavior of a uniprocessor by using parallel hardware, there is a defined order in which the blocks must be confirmed. We want to start execution of a given block early enough so that its results are ready by the time that block is reached by the block counter.

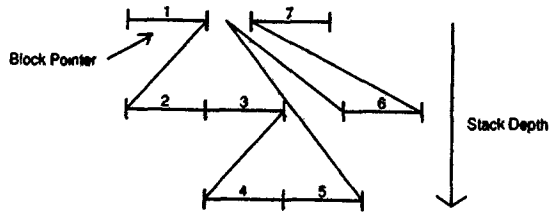


Figure 3. The call tree of a process provides heuristic data to guide the block starting process. Here, we want to start execution of blocks 2 and 3, while the block counter is at block 1, because these are the next sequential blocks We also want to start execution of block 6, because it is unlikely that confirming blocks 1–5 will side effect data which block 6 depends upon

But starting execution of a block earlier is potentially bad, because it will reference data which is older than necessary, and because the processor being used to execute it may be needed to execute a block which must be confirmed sooner.

These observations lead to the following block execution strategy:

1. Start executing blocks ahead of this block, or downward and to the left in the call tree.
2. Start executing blocks which branch downward to the right in the call tree.
3. If additional processors are required for (1) or (2), abort execution of the furthest right branches of the call tree, and reclaim their processors.

Heuristic (1) is very similar to instruction pre-fetching on conventional pipelined architectures. Since we are executing this block, we will next need to execute its successor.

Heuristic (2) depends on the fact that parallel execution of evaluations from the same level of the call tree are often independent, and can often be executed without conflicting side effects.

Heuristic (3) is a simple consequence of the fact that we must finish execution of the bottom left portion of the call tree prior to executing the right portion of the call tree. Thus, we are better off using the processors executing blocks to the right for more immediate needs.

These heuristics give us a very good handle on the difficult problem of resource allocation in the multiprocessor environment.

Treatment of Conditionals

The presence of conditionals in the language, however, makes the tidy resource allocation scheme described above much more difficult. Since we don't know which of the two paths a conditional will follow until the data computation is confirmed, we must start blocks executing down both branches, or risk delaying execution waiting for a block on the path not followed. Fortunately, we are free to start executing down both paths, since until we confirm a block, it performs no side effects. The blocks started along the unfollowed branch of a conditional are simply abandoned without confirming their side effects.

Allocation of processor resources in the presence of conditionals is substantially more difficult, since we cannot, except heuristically, predict which of two paths is more worthwhile for expending processor resources.

Loop Compilation

Loops are a special case of conditional execution. As with most parallel processors, there is an advantage in knowing the high level semantics of a loop construct. Rather than attempting to execute repetitive copies of the same block, incrementing the loop index as a setup for the next pass, it is far preferable to know that a loop is required, and that the loop is, for example, indexing over a set of array values from 0 to 100. We can then execute 101 copies of the loop block, each with a unique value of the loop index. The unpalatable alternative is to rely on the side effect detection hardware to notice that the loop index is changed in each block iteration, resulting in an abort of the next block on every loop iteration.

Unlike most of the parallel proposals for loop execution, however, the architecture we propose can also handle the hard case where the execution of the loop body modifies the loop control parameters, perhaps in a very indirect or conditional way.

Predicted Values

The way we deal with both conditional execution and with loop iteration involves *prediction* of the future value of a variable. In a sense, the dependency list maintenance already described is a form of value prediction: we predict that the value we are depending upon will not change by the time the executing block commits.

We can extend this idea to one which predicts the future value of variables likely to be modified from their current value prior to executing the block. One application of this technique is to the problem of loop iterations. Instead of relying on the current value of the loop index as the predicted value for the next iteration, we really want to predict the future value – predicting a different future value for each parallel body block we start.

Similarly, we can implement conditionals by predicting the value of the predicated result slot – predicting that it is true in one arm of the conditional, and that it is false in the other. The predicate is then evaluated, and when it confirms, it side effects the predicate result slot, aborting one of the two branches. When aborting the execution of conditional blocks, we must abandon execution of the block rather than update the variable and retry, as we would in normal block execution.

The Proposed Hardware

The hardware proposed for implementing this scheme is a shared memory multiprocessor, with each processor containing a two fully associative caches. The first of these, the *dependency cache*, usually holds read data copied from main memory. This cache also watches bus transactions in a way similar to the snoopy cache coherency protocol [7].

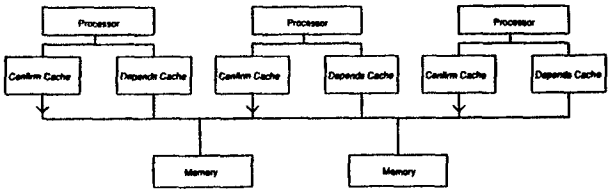


Figure 4. The processors share a common memory bus, and thus each cache see all writes to main memory The *depends cache* acts as a normal data read cache, but also implements the dependency list and predicted value features The *confirm cache* is used to allow processors to locally side effect their version of main memory, without those changes being visible to other processors prior to their block confirmation

A second cache, the *confirm cache*, holds only data written by this processor, but not yet confirmed.

Each main memory location, therefore, can have two entries in a processor's caches, one in the dependency cache, and one in the confirm cache. This is because we must maintain a knowledge of what data the processor's current computation has depended upon, in the dependency cache, while also allowing the processor to tentatively update its version of memory, in the confirm cache. For processor reads, priority is always given to the contents of the confirm cache if there is an entry in both caches, because we want the processor to see its own modifications to memory, prior to them being confirmed.

The Dependency Cache

The dependency cache performs three functions:

- It acts as a normal read data cache for main memory data.
- It stores block dependency information.
- It holds predicted values of variables associated with this block.

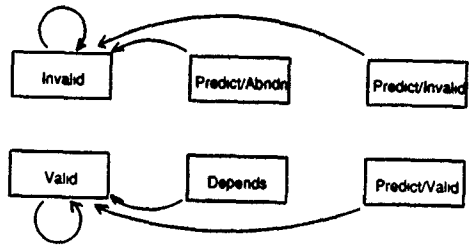
Figure 5 shows a state transition diagram for the dependency cache. There are six states shown in the cache state diagram: INVALID, VALID, DEPENDS, PREDICT/VALID, PREDICT/INVALID and PREDICT/ABANDON.

INVALID is the state associated with an empty cache line.

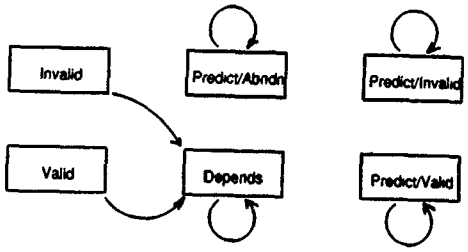
VALID is the state representing that the cache holds a correct copy of main memory data

DEPENDS indicates that the cache holds correct data, and that the ongoing block computation is depending on the continued correctness of this value.

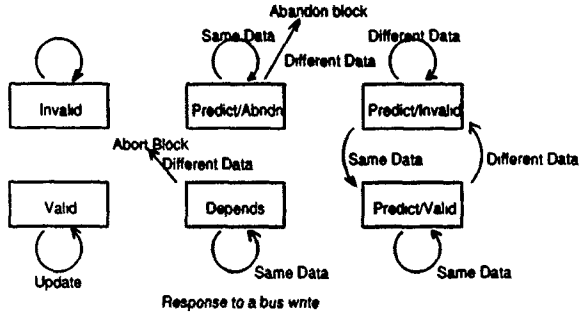
PREDICT/VALID indicates that the block is predicting that, when it is ready to confirm, the value held in this cache location will continue to be equal to the value in main memory. This state indicates that the held value is now



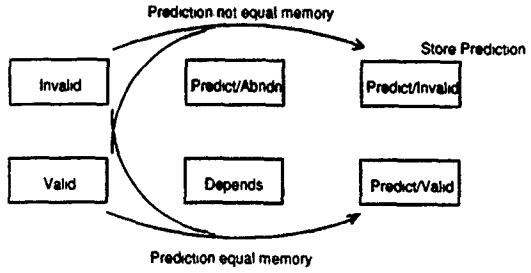
Response to initializing the cache when starting a new block



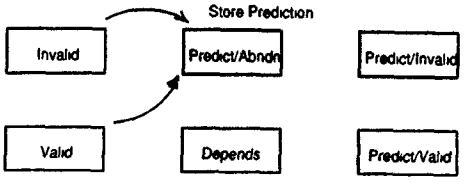
Response to a processor Load Request



Response to a bus write



Response to processor Predict operation



Response to a processor Predict/Abandon request

equal to main memory. This state differs from DEPENDS only in the action taken when a bus cycle modifies this memory location.

PREDICT/INVALID indicates that the block is predicting that, when it is ready to confirm, the value held in this cache location will be equal to the value in main memory. The contents of main memory currently differs from the value held in the cache.

PREDICT/ABANDON indicates that the block execution is conditional on the eventual contents of memory being side effected to be equal to that held in the cache. If it is side effected to some other value, the block will be aborted and not restarted (abandoned).

At the start of each block execution, the dependency cache is initialized. This results in setting all locations to either the INVALID or VALID states. Cache entries which are currently VALID, DEPENDS, or PREDICT/VALID are forced to the VALID state. All others are forced to the INVALID state.

Each time the processor loads a data item into a register from the depends cache, the state of the cache line is potentially modified. If the cache entry for the location is the INVALID state, a memory bus read request is performed, a new cache line allocated, the data stored in the depends cache, and the line state is set to DEPENDS. If the cache line is already VALID, then the state is simply set to DEPENDS. In all other states, the data is provided from the cache, and the state of the entry is unmodified.

A cache miss is allowed to replace any entry in the cache which is in the INVALID or VALID states, but must not modify other entries. We require this because the dependencies are being maintained by these states. This implies that the cache must be fully associative, since otherwise a conflict over the limited number of sets available for a given location would make it impossible to store the dependency information.

A Predict-Value request from the processor stores a predicted future value for a memory location into the depends cache. The cache initiates a main memory read cycle for the same location. If the location contains the predicted value, the cache state is set to PREDICTE/VALID. If the values disagree, the cache state is set to PREDICT/INVALID.

A Predict-Abandon request from the processor stores a predicted future value of a predicate into the depends cache, and forces the cache line to the PREDICT/ABANDON state.

A bus write cycle, taken by some other processor, can potentially modify a location in main memory which is held in the depends cache. Since there is a shared memory bus connecting all of the caches, each cache can monitor all of these writes. For INVALID cache entries, nothing is done. For VALID entries, the newly written data is copied into the cache, and the cache maintains validity.

For DEPEND entries, the cache is updated, and if the new contents differs from the old contents, the running block on the cache's associated processor is aborted and then restarted. This is the key mechanism for enforcing inter-block sequential consistency.

Bus writes to locations which are PREDICT/VALID or PREDICT/INVALID are compared with the contents of the depends cache. If the values agree, the cache state is set to PREDICT/VALID. If they disagree, the state is set to PREDICT/INVALID.

Bus writes to locations which are PREDICT/ABANDON are compared with the contents of the depends cache. If they agree, no change occurs. If they disagree, the associated processor is aborted, and the block currently executing is permanently abandoned.

The Confirm Cache

The confirm cache consists of a fully associative cache which holds only side-effected data. When the block is initialized, the confirm cache is emptied by invalidating all its entries. When the processor performs a side-effecting write, the write data is held in the confirm cache, but not written immediately into main memory. There, it is visible to the processor which performed the side-effect, but to no other processors. The confirm cache has priority over the depends cache in providing read data to the processor. If both hold the contents of a location, the data is provided by the confirm cache. This allows a processor to modify a location, and to have those modifications visible during further computation within the block.

Confirming

When the block counter reaches a block associated with a particular processor, and the block has completed execution, it is time to confirm that block. One final memory consistency check is performed. The load dependencies are necessarily satisfied, because if they were not, then the block would have been aborted. But the predicted-value dependencies may not be satisfied. We check these dependencies by testing if there are any entries in the depends cache which are in the PREDICT/INVALID state. Any entry in that state indicates a memory location whose contents was predicted to be one value, and whose actual memory contents are another. We must re-execute this block with the now-current value.

After performing this final consistency check, the side effects associated with this block may be performed. This operation consists of sweeping the confirm cache, and writing back to memory any modified entries. The write-back of these entries may, of course, force other processors to abort partially or completely executed blocks, if they have depended on the old values of these locations.

One consequence of using this cache strategy for implementing the dependency checking is that each block can perform multiple side-effects, and those side-effects will be visible only to the executing block until the block is confirmed. This relaxes the old requirements of exactly one terminal side-effect per transaction block, and allows the compiler flexibility in choosing the optimal block size independent of how many side-effects are present within the block.

Another important feature of using the cache as a dependency tracking mechanism is that when a transaction aborts, the valid entries in the cache remain, so that the re-execution of the block will likely achieve an almost 100% cache hit rate, reducing memory bus traffic and improving processor speed.

Consing Is Not A Side Effect

The Lisp operation *cons*, although it performs a write into main memory, is not a side-effect. We are guaranteed that no one else has a pointer to the location being written, and thus no one can be affected by its change. In the block execution architecture, this can be implemented by providing each processor with an independent free pointer Consing and other local memory allocating writes done within a block are performed using the free pointer. The value of

the pointer is saved when the block is entered, and abort resets the pointer to its entry value, automatically reclaiming the allocated storage. During a confirm, the free pointer's value is updated. All of these free pointer manipulations happen automatically if it is treated simply as another variable whose value can be loaded and side-effected. Writes of data into newly consed locations need not be confirmed. They can be best handled with a write-through technique in the depends cache. It is important that the depends cache not contain stale copies of data which has been written with a Cons write operation.

Introduction Of Explicit Parallelism

The addition of one primitive to the processor instruction set allows the re-introduction of explicit, programmer visible parallelism into the language supported by this architecture. The primitive performs a load operation without adding the location being loaded to the dependency list. In the depends cache, it returns Valid data if present, and reads memory if necessary to produce valid data. It never changes the state of the cache to Depends.

With this primitive, it is possible to express algorithms which compute results based on potentially stale data. In those cases where this is acceptable from an algorithmic point of view, then parallel execution with no inter-block dependencies can be supported. One example is the Gauss-Jacobi parallel iterative matrix solution technique, as contrasted with the sequential Gauss-Seidel technique.

Future Work

The independence of blocks is a strong assumption. Many situations arise, particularly in the area of argument passing and value returning, where it is awkward to require values to be passed in memory. There seems to be a requirement for a way to chain blocks together in a way such that the aborting of one block forces the aborting of a selection of dependent blocks. Some complex issues also surround the correct execution of conditionals.

There are dozens of parameters, such as the desirable block size, and the number of side effects per block, whose range of appropriate values must be established. There are alternative hardware implementations, involving, for example, dependency hash tables rather than fully associative caches, whose effectiveness must be studied. The strategies for allocating processor resources appear to be quite complex, when, for example, it is allowable to abandon a partially executed block to work on one closer to being confirmed.

While we currently believe that this work will lead to an architecture which can achieve factors of 10-100 times the performance of uniprocessors on unmodified Lisp programs, we have no proof, even in simulation. With modified programming techniques, eliminating gratuitous side effects, we may be able to achieve even higher speedups.

Relationship to other work

The intentionally evocative terminology of transactions, confirms, and aborts in this paper was chosen to reflect the close analogy of this technique to work in the database field. In particular, the work by Kung and Robinson on optimistic concurrency [8] is very similar, except for the inter-block ordering constraint. Taking the analogy of main memory as a database seriously may lead to other interesting ideas involving, for example, nested transactions. Similarly, the idea of supporting database transactions with dependency caches also looks like an attractive direction for further research.

The idea of splitting programs up into functional blocks terminating in side effects also occurs in the program proof literature, as in Crocker's work [9].

If one views the blocks of this scheme as very complex instructions, then this architecture devolves to a pipelined architecture for executing those instruction, and the interlocks to prevent inter-instruction dependency violations.

Morris Katz at M.I.T. has independently arrived at a similar programming methodology, but has concentrated on software implementations of the dependency tracking mechanisms [10].

Summary

This paper presents some suggestions on how one might use a small amount of hardware in the memory subsystem of a shared memory multiprocessor to enforce correctness of execution of parallel blocks of side effecting code. While the programs are being executed in a multi-processor environment, the programmer need not be aware of this fact. The architecture is capable of executing unmodified lisp code at a speedup dependent on the frequency of side effects. As the frequency of side effects is reduced, the effective execution rate will increase. Many issues remain in the design of an architecture for these mostly functional languages.

Acknowledgments

These ideas are a direct result of discussions with many employees of Symbolics and many M.I.T. students and faculty. Todd Matson, Ramin Zabih, David Chapman, Bruce Edwards, Alan Bawden, David Moon, Scott Wills, Daniel Weinreb, and David Gifford have been especially helpful.

References

1. Backus, J., "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs" *Communications of the ACM* Vol. 21 no. 8 pp. 613-641 (August 1978).

2. Henderson, Peter, "Is It Reasonable to Implement a Complete Programming System in a Purely Functional Style?" Technical memo PMM/94, The University of Newcastle upon Tyne Computing Laboratory (December 1980).

3. Agha, Gul A., "Actors: A Model of Concurrent Computation in Distributed Systems" M.I.T. Artificial Intelligence Labaoratory Technical Memo 844, (1985).

4. Abelson, Harold and Sussman, Gerald Jay, *Structure and Interpretation of Computer Programs*, M.I.T. Press, Cambridge, (1985).

5. Halstead, Bert, "MultiLisp: A Language for Concurrent Symbolic Computation" ACM Transaction on Programming Languages and Systems, (1985).

6. Moon, David A., "The Architecture of the Symbolics 3600" The 12th Annual International Symposium on Computer Architecture pp. 76-83 (1985).

7. Goodman, James R., "Using Cache Memory to Reduce Processor Memory Traffic" The 10th Annual

International Symposium on Computer Architecture pp. 123-131 (1983).

8. Kung, H.T. and Robinson, J.T., "On Optimistic Methods of Concurrency Control" ACM Transactions on Database Systems Vol. 6, No. 2, (June, 1981).

9. Crocker, Steven, "State Deltas: A Formalism for Representing Segments of Computation" USC-ISI Technical Report ISI/RR-77-61 (September 1977).

10. Katz, Morris J., "Paratran, A Transparent, Transaction Based Runtime Mechanism for Parallel Execution of Scheme" forthcoming S.M. thesis, MIT Department of Electrical Engineering and Computer Science Department, (May, 1986).